

Exhibit D

Diligence

Exhibit D1

Estream 1.0 Planning Document

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Completed	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
ApplInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	Bob		Status TBD	TBD	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLIM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	Igor/Amit	Not Done	Not Done	TBD	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	
LSM + Plug in	Anne	Done	Done	1 wk	
Client UI	Anne	Done	Done	1 wk	
Client Installer	Anne	Done	Done	1 wk	
Start Client	Anne	Done	Done	1 wk	
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	Curt	Status TBD	Status TBD	2 days	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

ECM (RAM disk cache) and EFSD executes a local "himom" executable
 Photoshop is installed locally and successfully executed from estream sets and applinstallblk produced by builder
 App Server and EMS integrated to copy "himom" executable using a dummy client
 App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
 office is installed locally and successfully executed from estream sets and applinstallblk produced by builder
 App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
 Following applications built and tested with local installation

Adobe Premier

Macromedia Director and Shockwave

Corel Suite

Lotus Suite

Photoshop is installed by AIM and executed from estream sets on App server

No Subscription

No License Management

RAM cache for ECM

Installation of Photoshop using AIM

Photoshop is installed by AIM and executed from estream sets on App server

No Slim Server

Disk based cache for ECM

Estream includes initial prefetched pages and these pages are prefetched during installation
 Fully functional estream bits (includes initial prefetched pages)
 Client software is run as a service
 App Server is started by Monitor
 Admin UI to stop and start app Server
 Application subscription from web server
 installation on client after subscription

Testing environment is setup (configuration of 3 servers and one client)

Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick
 Slim Server
 http protocol
 CNI supports unique message ids for NAD
 Fully functional LSM
 Real Accesstokens
 Uninstall applications
 Anti-Piracy support
 AppServer and SlimServer fail-over
 File spoofing

Clean builds by integration (George) (Raj will drive this)

Office is running with full functionality

Restructuring of client so it can be started at boot time
 Performance tuning
 Improve robustness
 application upgrade
 Crash resiliency
 All software purified and memory leaks eliminated
 (May be) Applets for monitoring server components

Office is removed from desktop of at least one person and reinstalled using estream

Code Freeze

Engineer

Server
 Sameer
 Mike
 Bhaven
 Amit
 Jae Jung
 Chungying Chu

Builder

David
 Bob
 Sanjay

Client

Dan
 Curt
 Anne
 Nick
 Raj
 Ameet

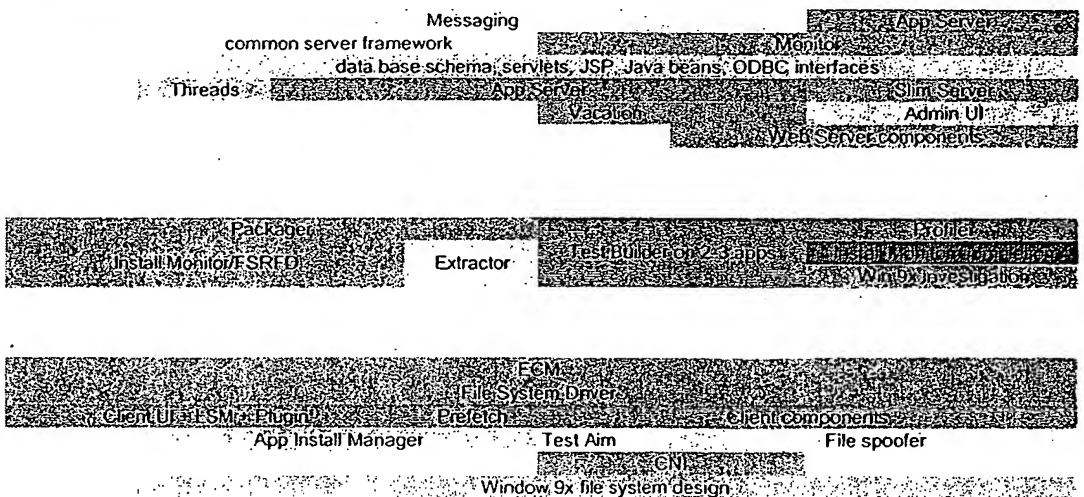


Exhibit D2

eStream Server Component Framework

Low Level Design

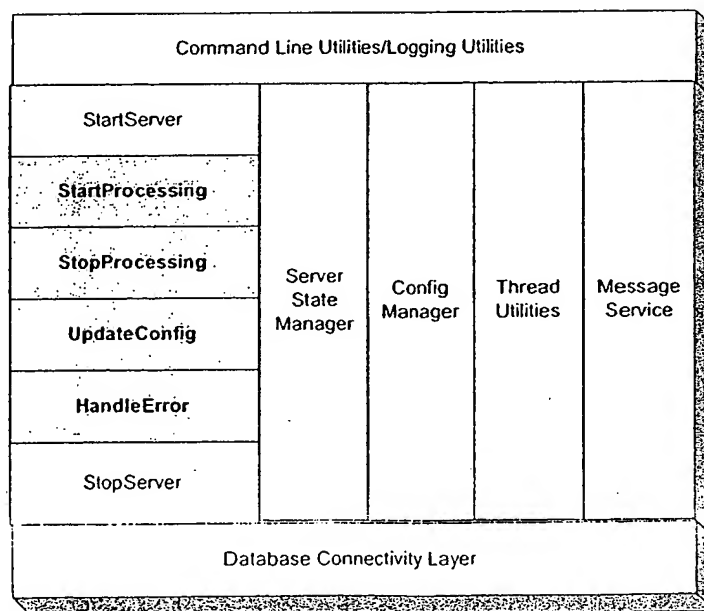
Michael Beckmann

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework*. To make use of the *Server Component Framework*, a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component “plug-able” within the framework.



eStream Server Component Framework Low Level Design

The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

Server State Manager:

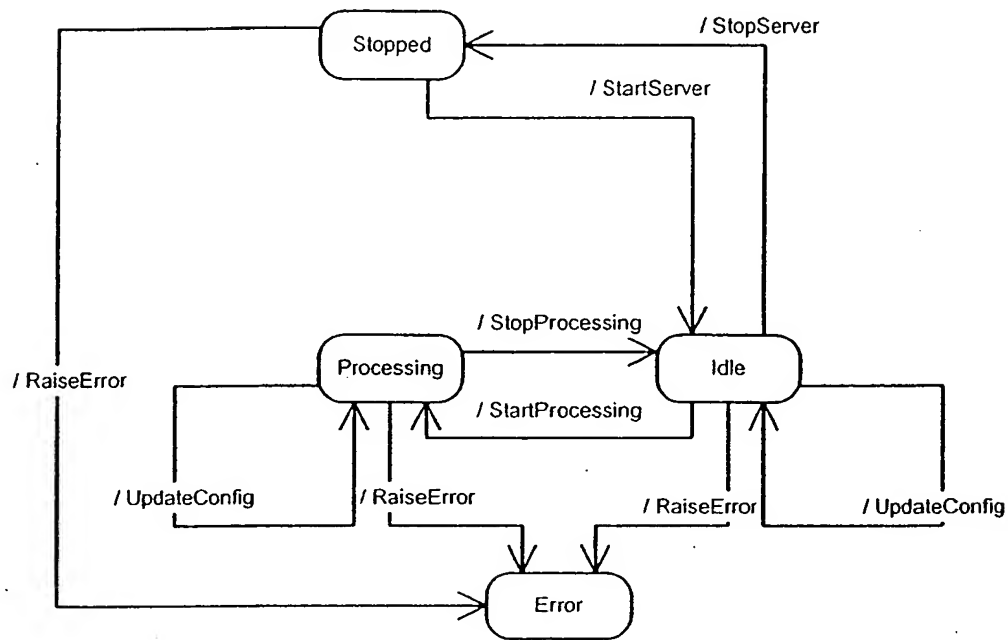
At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:

eStream Server Component Framework Low Level Design



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

eStream Server Component Framework Low Level Design

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:

word.exe windows2000sp3
excel.exe win98sp4

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager*. All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	Supports Dynamic Config	State	Description
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: <ul style="list-style-type: none">▪ Primary Monitor

eStream Server Component Framework Low Level Design

			<ul style="list-style-type: none"> ▪ Backup Monitor ▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: <ul style="list-style-type: none"> ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention. This info is consumed by the System Monitor.
HeartBeat-TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.
HeartBeatRate	Yes	Any State	Frequency at which the heart-beat is sent to this server component. Specified in milliseconds. This item is consumed by the System Monitor.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are

	case sensitive
--	----------------

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dsn=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optionally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
 - 4-Low : A warning which can be ignored.
 - 3-Medium: A warning which needs to be looked into.
 - 2-High: Recoverable Error in the component.
 - 1-Critical: Fatal Error. Needs admin assistance.

eStream Server Component Framework Low Level Design

- Logging level should be configurable. The following levels are to be supported.
 - 0: Only errors will be logged. This will be the default level.
 - 1: Errors and Warnings to be logged.
 - 2: Errors, Warnings and Debugging information to be logged.
 - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
 - Any existing <logfile>.bak will be deleted from the system.
 - The current <logfile> will be backed to <logfile>.bak
 - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

```
[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]
```

An example of this log format would be:

```
*****
Omnishift eStream Application Server
Server Started.
StartTime: [REDACTED] 16:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*****

[REDACTED] 16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[REDACTED] 16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[REDACTED] 16:31:19 -0700] 0 1-Critical Shutting down the server.

*****
Omnishift eStream Application Server
Server Stopped.
StopTime: [REDACTED] 16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*****
```

Format of Access Log Message:

[HEADER]
 [TimeStamp] [Thread ID] [Message]
 [FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with AutoReStart and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.

START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.
RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```

FSMTableEntry ServerStateMgr::FSMTable[] =
{
    { START, {{START_SERVER, STOPPED, START_SERVER, NULL},
              {START_PROCESSING, STOPPED, START_PROCESSING, NULL},
              {NULL_REQUEST, NULL_STATE, NULL_REQUEST, NULL}} },

    { STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST, NULL_STATE, NULL_REQUEST, NULL}} },

    { IDLE, {{START_PROCESSING, PROCESSING, NULL_REQUEST,
                &StartProcessing},
              {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
              {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
              {UPDATE_CONFIG, IDLE, NULL_REQUEST, &UpdateConfig},
              {NULL_REQUEST, NULL_STATE, NULL_REQUEST, NULL}} },

    { PROCESSING, {{STOP_PROCESSING, IDLE, NULL_REQUEST,
                &StopProcessing},
                  {UPDATE_CONFIG, PROCESSING, NULL_REQUEST,
                &UpdateConfig},
                  {STOP_SERVER, IDLE, STOP_SERVER, &StopProcessing},
                  {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                  {NULL_REQUEST, NULL_STATE, NULL_REQUEST, NULL}} },

    { ERROR, {{STOP_SERVER, STOPPED, NULL_REQUEST, NULL},
              {NULL_REQUEST, NULL_STATE, NULL_REQUEST, NULL}} },

    { NULL_STATE, {{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
                NULL}} }
};

```

Messaging Service Protocol:

eStream Server Component Framework Low Level Design

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	senderID	receiverID	targetID	Data
--------	----------	------------	----------	------

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	Description	Data
0x01	Request for current state	None
0x02	Acknowledgment	<ul style="list-style-type: none">▪ Current state▪ Load info▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
```

```
private:
    ServerState CurrentState;
    static FSMTableEntry FSMTable[];

public:
    ServerStateMgr(void);
    ~ServerStateMgr(void);

    ServerState SetState(ServerState);
    ServerState GetState(void);
    ServerState ProcessRequest(ServerRequest);
};
```

SetState	<p>Description: Sets the current state of the server component.</p> <ol style="list-style-type: none"> 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. <p>Note: SetState does not update the database directly as in the original design. The database is updated by the <i>System Monitor</i> once it has received an acknowledgement. A state transition is not complete until SetState returns successfully and the Monitor has update the database.</p> <p>Input: state value to set current state to.</p> <p>Output: current state after the new value has been set. If an error occurs will go to error state.</p> <p>Errors:</p> <ol style="list-style-type: none"> 1. Invalid state argument 2. Failure to either connect or commit state change to the database.
-----------------	---

GetState	<p>Description: returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state.</p> <p>Input: none.</p> <p>Output: returns the current state.</p> <p>Errors: None. Will always return a valid state.</p>
-----------------	---

ProcessRequest	<p>Description: request to the Server State Manager to change server state. This routine implements the guts of the state machine.</p> <ol style="list-style-type: none"> 1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to SetState to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized Han-
-----------------------	--

eStream Server Component Framework Low Level Design

	<p>dleError and transition to the ERROR state.</p> <p>Input: server transition request. Refer to table of valid requests defined above.</p> <p>Output: current state after the request has been completed.</p> <p>Errors:</p>
--	---

Server Component Framework:

<pre> class ServerComponent: ServerStateMgr{ // abstract base class private: ErrorInfo* Error; // maintains error if error was detected ServerConfig* Config; // holds common configuration Connection* Listener; // messaging utility public: virtual int StartServer(void); // may be specialized by a server component virtual int StopServer(void); // may be specialized virtual int StartProcessing(void) = 0; // must be specialized virtual int StopProcessing(void) = 0; // must be specialized virtual int UpdateConfig(void) = 0; // must be specialized virtual int HandleError(void) = 0; // must be specialized void Run(Request); } </pre>

StartServer	<p>Description: Called by the <i>Server State Manager</i> when a server component is to be started. The StartServer routine is provided as part of the <i>SeverComponent</i> class. It performs the following:</p> <ol style="list-style-type: none"> 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. <p>Note:</p> <ul style="list-style-type: none"> ▪ This routine must return immediately to the main thread. Otherwise the <i>Server State Manager</i> will be blocked. ▪ Successful return from the StartServer routine will put the server into the IDLE state. <p>Input: None.</p> <p>Output: Value of 0 if successful else error condition</p> <p>Errors: May return negative error condition</p>
--------------------	--

StopServer	<p>Description: Called by the <i>Server State Manager</i>.</p> <ol style="list-style-type: none"> 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process <p>Note: The monitor will update the database and perform logging.</p>
-------------------	--

	<p><u>Input:</u> None.</p> <p><u>Output:</u> Value of 0 if successful else error condition</p> <p><u>Errors:</u> May return negative error value</p>
--	---

StartProcessing	<p><u>Description:</u> Called by the <i>Server State Manager</i>. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers.</p> <ol style="list-style-type: none"> 1. Spawn a primary processing thread (also known as the boss thread). <ol style="list-style-type: none"> a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the <i>System Monitor</i>. <p>Note:</p> <ul style="list-style-type: none"> ▪ This routine must return immediately so that the <i>Server State Manager</i> can continue to operate in the main thread. ▪ This routine may make use of the <i>Server Configuration Manager</i> for obtaining specialized configuration information <p><u>Input:</u> None</p> <p><u>Output:</u> Value of 0 if successful else error condition.</p> <p><u>Errors:</u> TBD</p>
------------------------	---

StopProcessing	<p><u>Description:</u> Called by the <i>Server State Manager</i>. This routine must be defined by the specialized server component type.</p> <ol style="list-style-type: none"> 1. Reverse all actions performed by the StartProcessing routine. All worker threads should be joined or pooled in waiting state. <p>Successful return from this routine will put the server component into the IDLE state.</p> <p><u>Input:</u> None.</p> <p><u>Output:</u> Value of 0 if successful else error condition.</p> <p><u>Errors:</u> TBD</p>
-----------------------	--

UpdateConfig	<p><u>Description:</u> Called by the <i>Server State Manager</i>. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component.</p> <p><may require adding a new state to separate dynamic and static configurations></p> <p><u>Input:</u> None.</p> <p><u>Output:</u> Value of 0 if successful else error condition.</p> <p><u>Errors:</u> TBD</p>
---------------------	--

eStream Server Component Framework Low Level Design

HandleError	<p>Description: Component defined error handling routine to handle errors such as timeouts, etc.</p> <p>This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with the ServerComponent class.</p> <p>Input: None.</p> <p>Output: Integer value designating a handled error or failure. If the error cannot be handled then it is fatal.</p> <p>Errors: TBD</p>
--------------------	---

Run	<p>Description: This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the <i>System Monitor</i>.</p> <p>Note: The <i>Server State Manager</i> always runs in the main thread.</p> <ol style="list-style-type: none">1. Call ProcessRequest to transition the server component into the initially requested state.2. Enter main processing loop<ol style="list-style-type: none">a. Check for requests from the message service.b. Call ProcessRequest to service the request.c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. <p>Input: Initial Transition Request</p> <p>Output: None. This routine should never return</p> <p>Errors: None.</p>
------------	--

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
    ProcessRequest(Request);
    while (1)
    {
        Request = Listener->GetRequest();
        ProcessRequest(Request);
        Listener->AckRequest(Request, GetState, GetLoad, GetStatus);
    }
}
```

```
#include "ServerArgs.h"
#include "Server.h"
```

eStream Server Component Framework Low Level Design

```
int main(int argc, char* argv[]) {
    Args = new ArgList();
    Args->ProcessArgList(argv, argc);
    Server = new ServerComponent(GetValue(SID),
                                GetValue(DSN),
                                GetValue(DBUSER),
                                GetValue(PASSWD));
    Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
    private:
        char* Name;
        char* Value;
    public:
        NameValuePair();
        ~NameValuePair();
        char* GetValue(void);
        char* GetName(void);
        char* SetName(char*);
        char* SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
    char* Name;
    bool Required;
    char* DefaultValue;
    pFunc ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
    {"sid",          true,  0,      &ProcessSid},
    {"dsn",          true,  0,      &ProcessDsn},
    {"dbuser",       true,  0,      &ProcessDbUser},
    {"dbpasswd",     true,  0,      &ProcessDbPasswd},
    {0,             0,    0,      0}
};
```

```
typedef vector<NameValuePair*> ArgVector;
```

eStream Server Component Framework Low Level Design

```
class ArgList
{
    private:
        ArgVector          ArgVec;
        const ArgTblEntry* ArgTbl;

    private:
        NameValuePair*      ParseArg(char* Arg);
        char*               ParseName(char* Arg);
        char*               ParseValue(char* Arg);
        int                 ProcessArg(NameValuePair*);
        int                 FinalizeArgs(void);

    public:
        ArgList(const ArgTblEntry*);
        int     ProcessArgList(char* argv[], int argc);
};
```

eStream Server Component Framework Low Level Design

ProcessArgList	<p>Description: Process the entire argument list. In a loop for each argument argv[] ...</p> <ol style="list-style-type: none"> 1. Call ParseArg passing in argv[]. 2. ParseArg passes the result to ProcessArg 3. After processing the entire argument list and exiting the loop call FinalizeArgs <p>Input: argv and argc as passed into main() entry point Output: integer value designating success or failure Error:</p>
ParseArg	<p>Description: Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value></p> <p>Input: Next char* argument on the list Output: NameValuePair. NULL will be returned in the event of a syntax error Error:</p>
ProcessArg	<p>Description: This routine performs the semantic analysis of an argument.</p> <ol style="list-style-type: none"> 1. Look up name in the ArgTbl 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called ArgVec list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the ArgTbl <p>Input: NameValuePair Output: Integer value designating success or failure (0 for success, positive integer for other errors) Error:</p>
ParseName	<p>Description: Verify that the Name part of the argument conforms to being alpha-numeric</p> <p>Input: char* Name part of argument Output: char* Name else NULL Error: None</p>
ParseValue	<p>Description: Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters</p> <p>Input: char* Value part of argument Output: char* Value else NULL Error: None</p>
FinalizeArgs	<p>Description: Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the ArgVec.</p> <p>Input: None Output: Success or Failure Error:</p>

Configuration Manager:

```

class Tuple {
    string name;
    Value value;
};

class Value {
    int type;
};

class StringValue: public Value{
    string value;
};

class TupleValue: public Value {
    vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
    ConfigArray Array;
public:
    ServerConfig(serverId, dsn, dbuser, dbpasswd); // Initialize from db
    ServerConfig(serverId, string filename); // To initialize from a file.

    ConfigArray* GetConfigArray(int serverId);
    Tuple* FindConfig(string Name);
    int Reload(void);
    Tuple* GetConfig(int serverId ,string Name);
};

```

ServerConfig	Description: Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array Input: Server Id, Data Source Name, Database User name, and database users password. Output: None Errors:
ServerConfig	Description: Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. Input: filename of flat-file configuration. Output: None Errors:

eStream Server Component Framework Low Level Design

GetConfigArray	<p>Description: Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database.</p> <p>Input: ServerId specifying which server to retrieve configuration for</p> <p>Output: Returns a vector holding the configuration or NULL</p> <p>Errors:</p>
-----------------------	--

GetConfig	<p>Description: Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database.</p> <p>Input: ServerId specifying the server to retrieve configuration for and Name of configuration item.</p> <p>Output: Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered.</p> <p>Errors:</p>
------------------	--

FindConfig	<p>Description: Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager.</p> <p>Input: Name of the configuration item.</p> <p>Output: Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration.</p> <p>Errors:</p>
-------------------	---

Reload	<p>Description: Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results.</p> <p>Input: None</p> <p>Output: integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error.</p> <p>Errors:</p>
---------------	--

Logging Utilities:

<pre> class LogManager { private: char* FileName; int MaxFileSize; char* ResourceFile; // message catalog file char* GetMessage(MsgNum, MsgStr) public: LogManager(ServerId,Size=10); LogMessage(MsgStr); LogMessage(ThreadId, MsgNum, MsgStr, ...); </pre>
--


```
};
```

LogMessage	<p>Description: Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp.</p> <ol style="list-style-type: none"> 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. <p>Input: Thread Id, Message Number, Message String, and variable number of arguments. Output: None. Error:</p>
-------------------	---

GetMessage	<p>Description: Routine returns a message string from the resource file for the message number specified.</p> <p>Input: Message number, C Locale text string.</p> <p>Output: Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in.</p> <p>Error: If an error occurs trying to get a message from the resource file, a message will be logged to the error log.</p>
-------------------	--

```
class ErrorLog: protected LogManager
{
private:
    LogLevel ErrorLogLevel;
public:
    ErrorLog(ServerId, LogLevel=0, Size=10);
    LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	<p>Description: Writes output to error log file.</p> <ol style="list-style-type: none"> 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of LogMessage to write the buffer out to the file. <p>Input:</p> <ol style="list-style-type: none"> 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. ...: variable list of arguments to be inserted into the message string per the format. <p>Output: None.</p> <p>Error:</p>
-----------------	---

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

<p>Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested.</p> <p>Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.</p>	<p>Dependencies:</p> <ol style="list-style-type: none"> 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
<p>Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.</p>	<p>Dependencies:</p> <ol style="list-style-type: none"> 1. Phase 1 2. Database connectivity 3. Messaging Service
<p>Phase 3: Integration Testing</p>	<p>Dependencies:</p> <ol style="list-style-type: none"> 1. Phase 2 2. System Monitor (including backup) 3. SLiM Server, App Server, Web-Server
<p>Phase 4: Stress Testing See section on stress testing for details</p>	<p>Dependencies:</p> <ol style="list-style-type: none"> 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a configtest.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlogtest.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing**, **StopProcessing**, **HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
 - a. data persistence.
 - b. detection capabilities and response.
 - c. auto restart.
3. Test to kill individual server component processes.
 - a. data persistence.
 - b. detection capabilities and response.
 - c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
 - a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional debugging information to special log files.

Open Issues

Exhibit D3

eStream Set Format Low Level Design

Sanjay Pujare and David Lin

Version 0.7

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. The eStream Set file size is limited to 2^{64} bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to EStreamSet
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVTOffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFOffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.

- **AppBaseNameIsAnsi** [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **AppBaseNameLength** [4 bytes]: Byte length of the application base name.
- **AppBaseName** [X bytes]: Base name of the application. I.e. "Word 2000". Null-terminated.
- **MessageIsAnsi** [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **MessageLength** [4 bytes]: Byte length of the message text.
- **Message** [X bytes]: Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries** [4 bytes]: Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber** [4 bytes]: Version number of the root directory.
- **FileNumber** [4 bytes]: File number of the root directory.
- **VersionNameIsAnsi** [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VersionNameLength** [4 bytes]: Byte length of the version name
- **VersionName** [X bytes]: Application version name. I.e. "SP 1".
- **Metadata** [32 bytes]: See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to Number-Files-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

SOFT entry structure: (variable number of entries)

- **Offset** [8 bytes]: Byte offset into CAF of the start of this file.

- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.

- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.
- **NumEntries [4 bytes]:** Number of entries in the directory. Some entries are used for storing long file names and some are unused due to deleted files. So the NumEntries must be equal or less than NumFiles.

Fixed length entry for each file in the directory consists of 2 formats (short format for storing files with name that fit the 8.3 convention; and long format for storing long file names). Each entry is 84 bytes and the entry are aligned on every 4K page boundry. Thus, in the first 4K page of the directory, the padding consists of 12 unused bytes (52 bytes for header + 48 entries * 84 bytes per entry + 12 unused bytes = 4096 bytes). In all subsequent pages, the padding is 64 bytes (48 entries * 84 bytes per entry + 64 unused bytes = 4096 bytes):

Short Filename entry:

- **Format [1 byte]:** Format of this entry, should be 's' for short format, 'l' for long filename format, or possibly 'u', for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [1 byte]:** Length of long file name.
- **UNUSED [1 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the short file name. Algorithm TBD.
- **ShortName [24 bytes]:** 8.3 short file name in unicode
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
 - **Bit 0:** Read-only – Set if file is read-only
 - **Bit 1:** Hidden – Set if file is hidden from user
 - **Bit 2:** Directory – Set if the file is an eStream Directory
 - **Bit 3:** Archive – Set if the file is an archive
 - **Bit 4:** Normal – Set if the file is normal
 - **Bit 5:** System – Set if the file is a system file
 - **Bit 6:** Temporary – Set if the file is temporary

The bits of the **eStream flags** have the following meaning:

- **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
- **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
- **Bit 2:** Read-only – Set if the file is read-only

Long Filename entry:

- **Format [1 byte]:** Format of this entry, should be 'l' for long filename format.

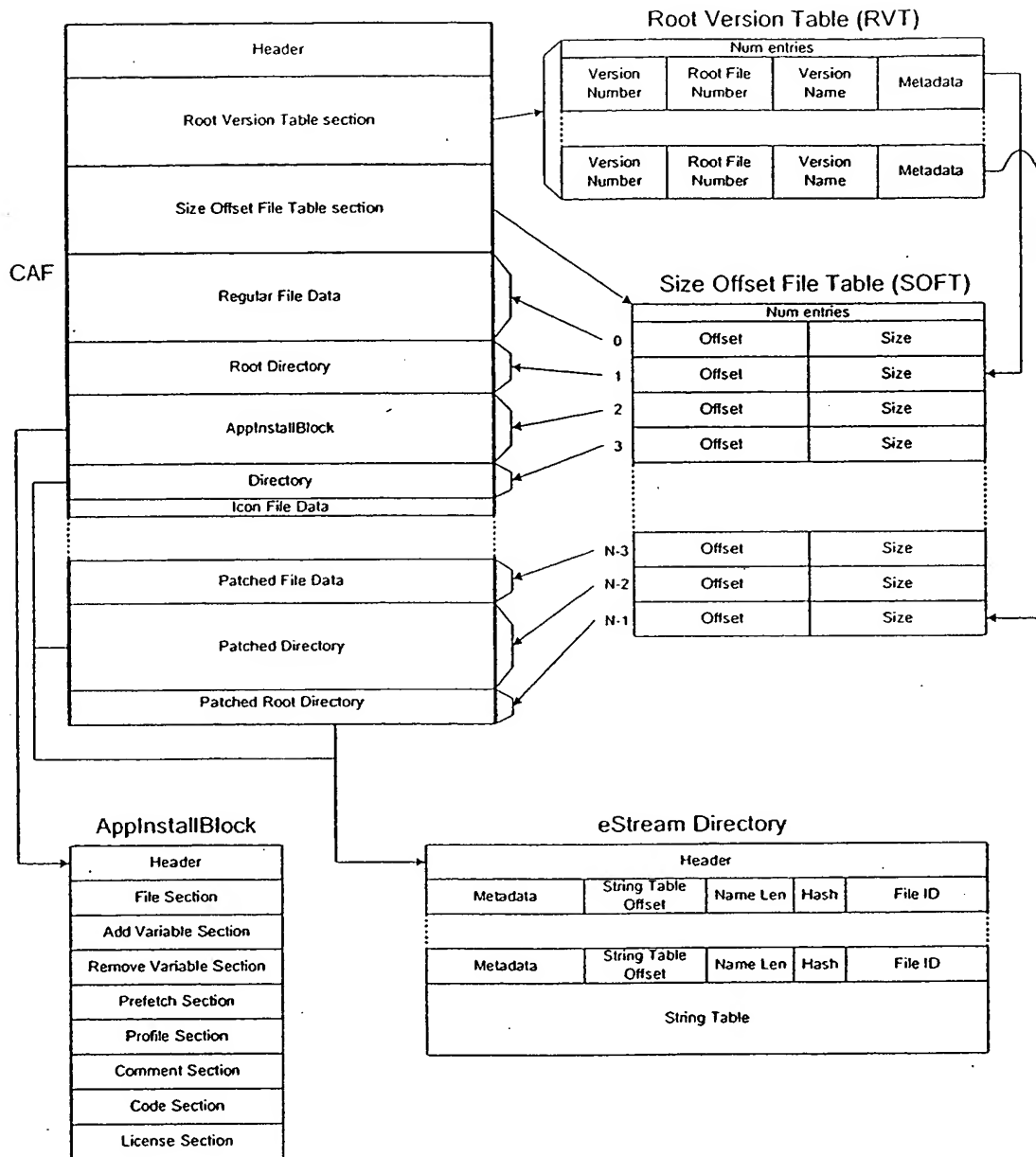
eStream Set Format Low Level Design

- **Index [1 byte]:** Number of this entry out of those used for this file's long name.
- **UNUSED [2 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the long file name. Algorithm TBD.
- **LongName [76 bytes]:** Long filename in Unicode format.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set



v 02

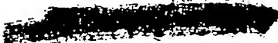
Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

Exhibit D4

eStream 1.0 High Level Design

Version 1.0



Introduction

This document describes the high level design for the eStream 1.0 product. It is essentially a summary and a tying together of the low level designs for each component in the system. The organization of this document is:

- ❑ Basic overview of the entire system
- ❑ Block diagrams for the client, server, and builder portions, showing all major components
- ❑ General discussion of each component, and pointers to the low level documents for these components
- ❑ A list of known issues

To understand the problem being solved in this design, see the “eStream Requirements Document” for information.

Note that this design is for a Windows NT4.0 and Windows 2000 client **only**. As work progresses on a Windows 95/98 client, the designs here will be updated.

Overview

eStream 1.0 encompasses the following basic features:

1. A distributed file system for application files, residing on a server and cached on a client.
2. A small client “player” program to allow local execution of applications that reside on the servers.
3. Authentication using tokens supplied by a license server to each active client.
4. A managed database of information about applications available to client machines, and subscription and usage data for each registered user.
5. Integration with service provider web servers to allow users to subscribe to apps and manage their accounts.
6. Monitoring of all servers to detect problems and allow automatic failover.
7. A build system that analyzes applications and enables them to be executed by the client and server.
8. Anti-piracy features to discourage unauthorized copying and use of subscribed applications.

As a way of overview, here are the processes that take place to enable and execute a Windows application from a client machine.

- ❑ The eStream builder is used to create an *eStream set* for the application. The application is installed on a clean machine, with the builder tools running. These will monitor all file installs and registry updates required to run the application, and encode them into a binary file—the eStreamSet—that will be installed on a service provider's eStream application server (app server).
- ❑ A user must download and install the eStream client (ECE) onto her machine, and register as a valid user from a service provider; this will be done using the service provider's web site.
- ❑ The user will subscribe to an application from the service provider; a browser module on the client machine will be notified and send a message to the ECE about this event.
- ❑ The ECE will communicate with the service provider's eStream license server (Slim server) to verify the newly subscribed app and all permissions, and will install a small portion of the application onto the client system—essentially, the registry entries, shortcuts, and small shared files necessary for execution.
- ❑ All application files that are not installed on the client will be accessed via a separate eStream file system (EFSD)
- ❑ The user will now see standard shortcuts for subscribed applications, exactly as though the app were installed locally.
- ❑ Starting an application, via a command line or double-clicking a shortcut, will cause the client machine to start executing the application on the EFSD. This means the virtual memory manager will request pages from the EFSD during page faults.
- ❑ These requests will be forwarded from the EFSD to the eStream cache manager (ECM), a component of the ECE, and on to the app server, assuming the page requested is not in the cache.
- ❑ Before any page request is fulfilled by the ECE, the client license subscription manager (LSM) will check that the user has permission to run the application, requesting an *access token* from the Slim server if an existing one has expired.
- ❑ This valid access token is sent from the client to the app server for every page request; this authenticates the request.
- ❑ The server monitor will be continually checking the state of the app servers and Slim servers. If any are down, it will take them off line.
- ❑ The client has a list of valid Slim and app servers for each registered service provider and subscribed application. If response time for any of these is bad, it will stop using it and fall back on the rest.

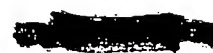
Block diagrams

The following are simple block diagrams of the client and server components. Some conventions:

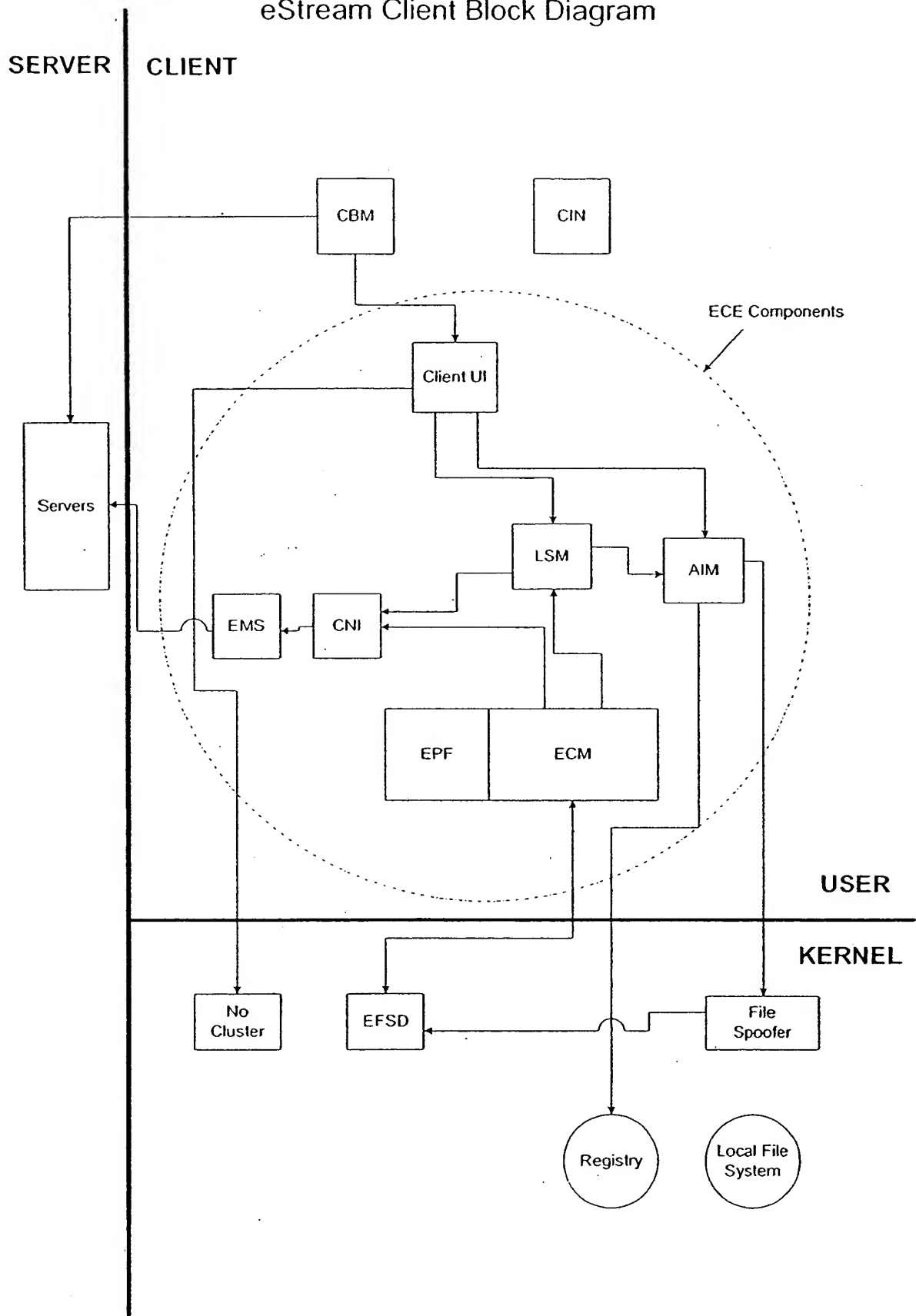
- ❑ A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.

- A line between components represents an interface call from one to another. If A calls B, there's an arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

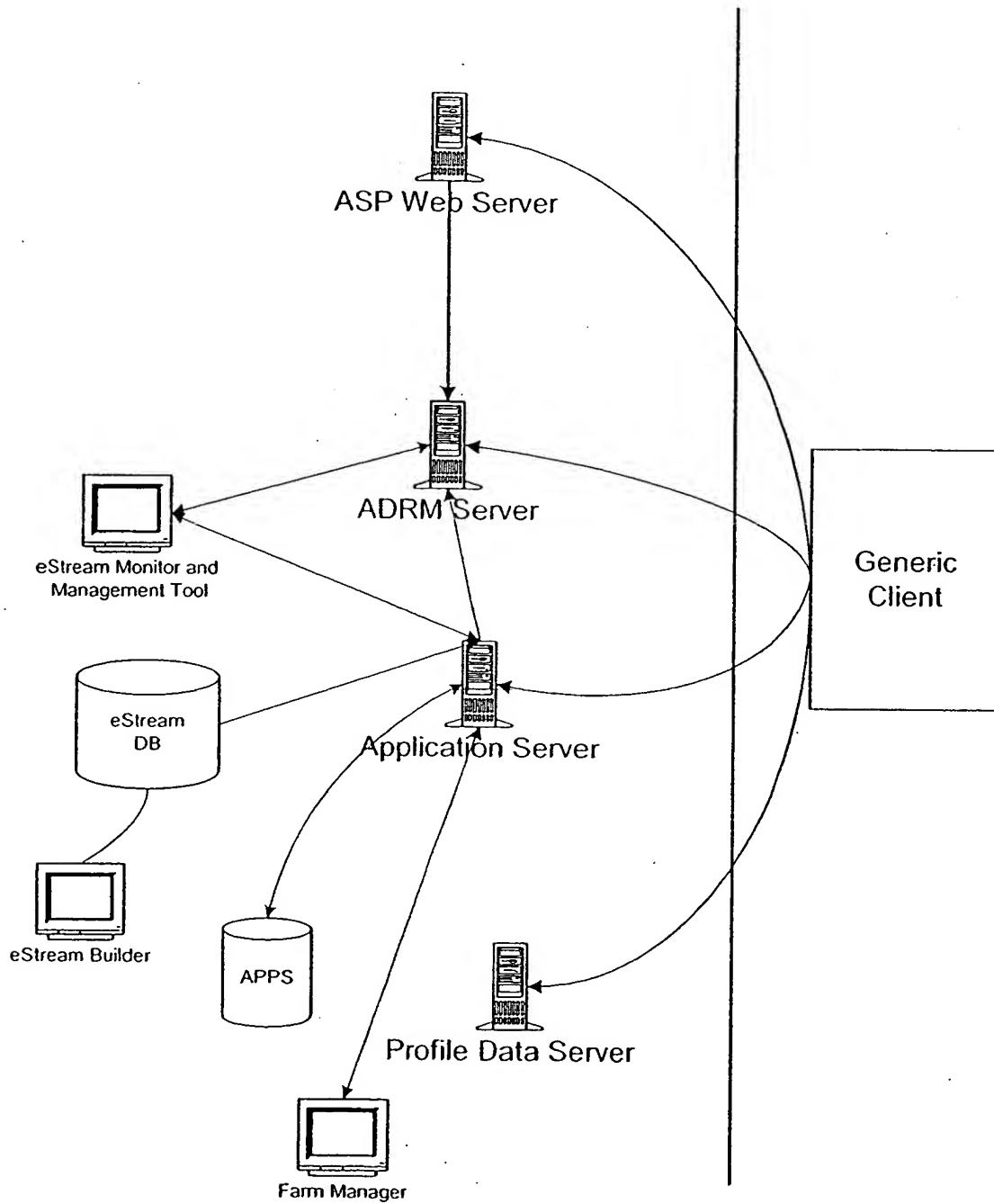
Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.



eStream Client Block Diagram

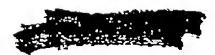


eStream Server Block Diagram



eStream Builder Block Diagram

???

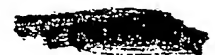


Component descriptions

Client components

The eStream client consists of the following components illustrated in the diagram above:

- ECE: the eStream Client Executable. This is the aggregation of several user space components into a single executable, operating as a Windows service.
- LSM: the License Subscription Manager (part of the ECE). This tracks and handles all required user information needed by the client: service providers, subscriptions, and access rights.
- AIM: the App Install Manager (part of the ECE). This is responsible for installing all necessary bits onto a client machine in order to run a subscribed application. It also uninstalls all local app bits when unsubscribing.
- ECM: the eStream cache manager (part of the ECE). This is the user-space component that handles requests from the EFSD, and manages the on-disk and in-memory cache of file contents.
- EPF: the eStream PreFetch component (part of the ECE). This works closely with the ECM to handle prefetches of pages for running eStream applications (as opposed to demand fetches, handled by the ECM).
- CNI: the Client Network Interface (part of the ECE). This manages queues of requests from various client components to the app and Slim servers.
- EMS: the eStream Message Service (part of the ECE). This library, used in both the client and servers, handles the actual network sends and receives between remote machines.
- CBM: the Client Browser Module. This is a client-side web browser plugin that is used to handle notification from a service provider's web server to the ECE, when user updates have taken place.
- CIN: the Client Installer module. This small component installs, upgrades, and uninstalls all the required client software.
- FSP: the File Spoofer. This is a kernel-mode driver that is used to redirect requests, intended for local filesystems, to the EFSD. It is a file system filter driver that sniffs all Create requests to the necessary local FSDs, compares the filenames with a list of files that must be spoofed, and if a match is seen, redirects the request to the EFSD.
- EFSD: the eStream File System Driver. This is a standard Windows NT FSD, handling all necessary FS requests from the I/O Manager. It ultimately sends these requests to the ECM to be satisfied (either locally or remotely).
- No Cluster. This is a kernel-mode driver that simply disables file system page clustering for threads running as part of eStreamed applications.



ECE

The ECE is the Windows service that comprises the bulk of the user-space eStream client software. It provides an overall main program loop, as well as the user interface component for all client components that must communicate with a user.

LSM

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs Slim servers to get updated lists of subscribed apps.

AIM

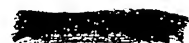
The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to them, and it gets requests from the Client UI to uninstall applications.

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers to get the AppInstallBlock. The AIM uses the AppInstallBlock to then make the appropriate calls to the file spoofer; to install some files on the local disk; to “warm” the cache and to update the start menu and other short cuts as needed.

ECM

The ECM is part of the ECE. Its goal is to:

- Handle all file requests from the EFSD, either by using previously cached contents or requesting the contents from a server.



- Work with the LSM to insure that all applications have appropriately validated licenses before their files are accessed.

The ECM handles the volatile and non-volatile eStream cache on the client machine. It performs demand fetching from the appropriate server(s). Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

EPF

The ECM is part of the ECE. Its goal is to intelligently use prefetching of file data to reduce latency of pages requested from the EFSD; this prefetching may result from profiling data or heuristics.

CNI

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

EMS

CBM

CIN

The client installer is a simple InstallShield (or simpler) application that will install all of the required client software.

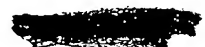
FSP

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the EFSD, or to another, non-eStream'ed file.

EFSD

The EFSD provides standard kernel file system interfaces to the I/O manager and other kernel-mode components. It works with the NT Cache Manager to efficiently cache file



and directory contents. Its view of the ECM is essentially like that of a disk driver, sending primarily read and write requests as needed.

No Cluster

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Server components

The following are the server components for eStream 1.0:

- ❑ App Server. This is essentially a file server for eStream sets. It satisfies requests for pages from eStream files from the client.
- ❑ Slim Server. This handles requests from a client for user and service provider information, and grants access tokens to the client for executing eStream applications.
- ❑ Web Server. ???
- ❑ Monitor. This enables an administrator to view the server components. It regularly pings the various servers, takes disabled ones off line, and adds new ones to the pools.
- ❑ eStream Database. This tracks all user information and server resources for a given service provider.

App server

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Slim server

The Software License Management (Slim) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

ASP web server

This describes only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the Slim servers to gather the statistics and display them.

Database

Builder components

These are the builder components for eStream 1.0:

- ???

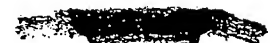


Exhibit D5

eStream Web Server Load Monitoring Applet Low Level Design

Jae Jung



Functionality

One of the requirements for the eStream web server is a facility for monitoring server load (eStream requirements 3.0, 3.2, 3.4, 3.5). Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the eStream web server installation.

The load-monitoring applet will present information in the following formats through a graphical interface:

- Real-time server load information plotted on strip chart
- Historical server load information plotted on line chart
- Multi-server real-time or historical load information on a line chart

Requirements

The following list details the provisional requirements for the load-monitoring applet. The remainder of this design document is based on these requirements.

1. The applet will be able to display server loads in "real-time" as load data is retrieved from the server.
2. The applet will be able to display (in a separate mode from real-time monitoring) historical load information to the extent that this information is available in the database.
3. The applet will be configurable (via applet parameters) for the following settings:
 - Data retrieval rate, i.e., the frequency with which the applet request new load data from the web application server
 - Chart window size, i.e., the number of datapoints shown in the chart window at any one time.
4. The applet will be capable of concurrently displaying the load of multiple servers in a clear and concise fashion.
5. The applet will support the displaying of cumulative and average load statistics for multiple servers.

6. In real-time mode, the applet will operate as a strip chart, with the fastest chart speed determined by a global configuration setting in database, typically corresponding to the frequency with which the eStream Monitor inserts load records into the database.
7. The applet will retrieve load information from the database via an http connection to the eStream web app server.
8. The applet will run in browsers with Java 1.0.x and 1.1.x support.
9. Internationalization support. Both the Applet and the backend pieces should be internationalizable.

Description

The load monitoring applet is comprised of two components. The first component will manage the retrieval of real-time or historical server load information from the eStream database. The second will consume this data and present this information to the user in a clear and concise graphical format. In addition to the applet, server-side objects will need to be written or extended to service data requests from the applet.

For the alpha-release of the eStream web server, the graphical presentation component will not be written internally; rather a commercially available applet or package will be used. Other aspects of the applet and the server-side components will be implemented to facilitate transitioning to an internally developed graphical component if such a decision is made for later releases.

User Interface Design

The following two screen shots are representative of the way that the load monitoring applet might be used in a particular monitoring/administration Browser interface. The first shot shows a general server administration and monitoring UI and the second shows a detailed load monitoring UI within which the load monitoring applet will be embedded.

The UI options shown in the second shot illustrate some of the reporting options for which the load monitor will be initially configurable. The applet(s) will be readily extensible to generate additional reports and more complex data combinations if desirable.


BEST AVAILABLE COPY

Untitled Document - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Search Favorites History

Address http://localhost:8080/jstream/server/UserServlet



Server Administration - Server Monitor

Name	Type	State	Action	Server Log	Server Load
goofy	SLIM	Unknown		View	<input type="checkbox"/>
mickey	SLIM	Unknown		View	<input type="checkbox"/>
minnie	SLIM	Unknown		View	<input type="checkbox"/>
scrooge	SLIM	Unknown		View	<input type="checkbox"/>

[View Server Loads](#)

[View Server Loads \(Advanced\)](#)


Done

Untitled Document - Microsoft Internet Explorer

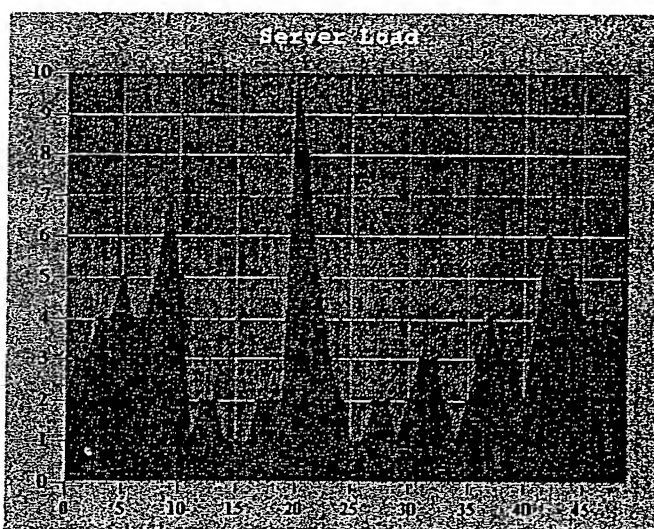
File Edit View Favorites Tools Help

Back Forward Stop Search Favorites History

Address http://localhost:8080/jstream/jsp/monitor/Client.htm



Server Load



Start Time: January 20 12 AM

Stop Time: January 20 12 AM

[View Current](#) [View Historical](#) [Refresh Chart](#)

Done

Interface Definitions

The load monitoring applet has two interfaces with other webserver components: <APPLET> tag parameters for its configuration within the web browser page and the HTTP request/response format with the web application server to request and receive load data.

1. Applet <-> HTML Page Interface (<APPLET> Tag Parameters):

Parameter	Req'd?	Significance	Default
hostName	Yes	Host name of webserver	None
hostPort	No	Host port of webserver	"80" (int)
chartSpeed	No	Time (in seconds) between chart scroll; also the resolution of the x-axis	"5" (int)
updatePeriod	No	Interval (in seconds) between HTTP requests for server load data. Note if chartSpeed < updatePeriod, the applet buffers load data for presentation. Maximum information latency will at most be equal to this value plus round trip time for the data request.	"10" (int)
chartWidth	No	Width (in ticks) of the applet chart; in conjunction with chartSpeed, implicitly determines the amount of data displayed	"50" (int)
mode	No	"current" for real-time monitoring and "history" for historical	"current"
startDate	No	if mode="history", the starting date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored.	none; value must be in Java Date format
stopDate	No	if mode="history", the stop date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored. Note that stopDate override the chartSpeed setting insofar as x-axis resolution is concerned.	none; value must be in Java Date format
numServers	Yes	number of servers to be plotted	1 (int, max 50)
serverName1 serverName2 ... serverName50	Yes	the id of the server(s) being plotted (up to 50 servers can be plotted at once). This must correspond to a existing serverID in the Server database table	none
serverData1 serverData2 ... serverData50	No	an initial set of data with which to display the initial chart. Note that these parameter(s) are the default means by which historical load data is displayed. Each set should be a comma delimited list of numbers (float)	none

- Note that applet tag parameters are all strings; where the applet does an internal type conversion, the target type format of the particular parameter has been noted.
- Note that the parameters determining the appearance of the chart (e.g., line colors, grid painting options, custom labels, etc.) are not included in this list. These parameters will either be determined by the parameters available for configuration in the commercial charting applet or TBD at some later date if the charting component is developed internally.

2. Applet <-> Web App Server Interface (HTTP)

Input:

The load monitoring applet will request load data from the webserver by executing the web server's MonitorServlet with the following parameters:

```
href="/MonitorServlet?action=getLoadData
    &serverId=...
    &startDate=...
    &stopDate=...
    &numPoints=...
```

where:

- serverId is a comma-delimited list of one or more server(s) for which load information is being requested. Note that this serverId corresponds to the serverID attribute in the Server database table.
- startDate is the (exclusive) starting date (in Java Date format) for which the load information is being requested
- stopDate is the (inclusive) end date (in Java Date format) for which the load information is being requested
- numPoints is the number of data points requested between the start and stop dates.

The applet will process the return data points as equally time-spaced points between the requested start and stop dates.

In addition, if startDate=stopDate, only one load data point (i.e., the most recent) will be returned by the servlet.

Output:

The load monitoring applet will expect load information from the web server via HTTP response in the following XML-like format:

```
<loadData>
```

```

        <serverid=a >
            <LOADLIST>
                <LOAD value=x1/>
                <LOAD value=x2/>
                <LOAD value=x3/>
                ...
                <LOAD value=xN/>
            </LOADLIST>
        </server>
        <server id=b>
            <LOADLIST>
                <LOAD value=y1/>
                <LOAD value=y2/>
                <LOAD value=y3/>
                ...
                <LOAD value=yN/>
            </LOADLIST>
        </server>
        ...
    </loadData>

```

In the above example, x1 to xN represents load values for the server a (by serverID), and y1 to yN represents load values for server b.

Testing Design

The load monitoring applet and related server-side Java code will need to be tested according to the plan outlined for other web server components in the Web Server/Database Low Level Design (WebServerDB-LLD.doc). Additionally, it should be noted that certain applet-related parameters (i.e., updatePeriod) that affect the frequency with which the applet requests load data from the web server are good candidates for tuning in order that a good balance between UI/measurement response and web server response performance be struck.

Upgrading/Supportability/Deployment Design

Upgrading/Supportability/Deployment Design will follow the model described for the Web Server in the Web Server/Database Low Level Design (WebServerDB-LLD.doc).

Open Issues

1. How much will it cost to deploy the chosen commercial java applet/package(s) as an OEM installation? We need to find this out before we decide on a commercial

package. Also, another criteria for the choice would be: will we get support. Do we get the source code too?

2. Longer term, where's the cost/benefit breakpoint for the above where it makes more sense to write our own charting applet/package?